Michael James Lukiman
Courant Institute of Mathematical Sciences

1.a. The greedy algorithm is sometimes beneficial, but it is not optimal: a counter example is a rod of Length 4 where the value list is $1 for length 1, $6 for length 2, $10 for length 3, and $5 for length 4. The greedy algorithm would select the most cost effective length 3 for the first subproblem and then length 1 (considering there is only 1 unit left), totaling $11 for maximization. The optimal way, on the other hand, using dynamic bottom-up look-up tables, is knowingly selecting two lengths of 2, which totals $12 for maximization.

1.b. [[1 2]] and [[2], [5]] would result in a total of 24 operations, while the standard approach results in 20.

2.a. It's not necessarily a subtree, so max height is the number of elements n, where each element is a child of the previous. So how do we count the height? The algorithm takes the child of the root of the subtree, adding to a counter, and then takes the child of that child until there are no more children. The counter's value at the end of this algorithm will be one of the possible heights. Then, take the most recent subtree with more than one child, and traverse it, adding on to the height recorded at that position. To store these possible heights, we hold an array of distance from root of each element and the height already calculated up to that point. And continue.

3.- Palindrome

Have two scanners, one coming from the beginning of the array [0] and one coming from the end of the char array [n-1] (we can reverse the word for this second array, adding n steps). For every character that shares a previous subsequence, as in our LCS algorithm, add a value

to the subsequence matrix from the values to the left and above it. At the end, the length and returned value will be the longest palindrome. Here, we compare character to retcarahc, with a matrix like:

		c	h	a	r	a	c	t	e	r
	r	0	0	0	1	1	1	1	1	1
	e	0	0	0	1	1	1	1	2	2
	t	0	0	0	1	1	1	2	2	2
	c	1	1	1	1	1	1	1	1	1
	a	1	1	2	2	2	2	2	2	2
	r	1	1	2	3	3	3	3	3	3
	a	1	1	2	3	4	4	4	4	4
	h	1	1	2	3	4	4	4	4	4
	c	1	1	2	3	4	4	4	4	5

The highest value diagonals in order will produce the longest palindrome.

4.a. A[1...n], real

We are finding the Si,j subsequence that is the maximum of the array's elements of indexes from i to j.

We start with the bottom case, where i = j <= n and they cycle through the array. These are the values of the array elements themselves. This takes no computation.

We then find the sum of each inclusive pair, which is the subset where i + 1 = j. We keep these in a n-1 size array, and this adds n-1 computations.

We then find the sum of triplets. The sum of triplets is when we can start to utilize the strengths of dynamic progamming. Triplet sums would be where i + 2 = j. We can do this by taking the nth

element from the pair sums and add A[n+2] for the third. This would take (n-1)+(n-2) computations.

With quartets, we would add every other pair in the pair sums array. This would be (n-1) / 2 in terms of additional space and computations.

With the pentuplets, we would use the same approach as triplets, taking the elements in the quartet sum array and adding A[n+4] of the original array.

This can be continued ad infinitum using a for loop with i through j, up until i + (n-1) = j.

In each of these, we use a find_max snippet to compare the result with the previous max, ending up at the global maximum of the set after adding all n elements.

As we can see, the required computations are at most n and fewer for each iteration, where the number of iterations is at most n itself. Thus, it is upper bounded by n^2.

4.b.

L[j] is the max we found through the above algorithm described, with j being variable. The recurrence relation is T ~= 3T(n/(i-j)), with three operations through n for any chunk size (i-j). Since we are trying to fill in L[1...n], we can use L[1,2] and save that, adding on L[3] and so on to L[n], taking O(n). The max of that series would be where max S is, though instead we would need to start from i, substracting, the saved sum from L[0] to L[i]. This would take O(2n).

4.c.

Without memorization, the O(n^2) algorithm would stay bounded, because the non-memorizing upper bound would be the running time of adding the pairs n times, give or take it being (n * (n-1)). The O(n) algorithm, on the other hand needs memory to prevent it from having to recalculate what comes before L[i], so it would then become polynomial if non-memorizing.

4.d.

The algorithm would stay more or less consistent, except that we would divide instead of subtract for elements in L before L[i], and that would would multiply instead of sum.

5.a. X[1...m] and Y[1...n] are two given arrays. A common supersequence of X and Y has Z[1..k] where X and Y are subsequences.

The length of Z is m+n, if X and Y are distinct exact subsets with no other subsets. Of course, that's not always the case. If they share subsequences, you can subtract the length of the longest common subsequence. M would include the elements of X and the elements of Y, but some of those elements would be the same. Thus we have a recurrence relation where each step would have n - 1 considerations. If X[i] = Y[j], then we fill a sequence matrix with the number of sequences in common up to that point.

For the base cases M[0,j], scanning based on X will have either 0 or 1 common subsequences. For M[i,0], there can be up to an i length subsequence, or perhaps 0 if no characters are alike.

5.b. We wish to fill in a subsequence matrix that recurs each element of X (0 through m) and compares it to Y (o to n), citing 1 if similar and 0 if not already having a 1 or more before it. This will in total

be a O(2mn) iteration loop. Iterating down again, wherever X and Y share subsequences, add to Z. For every cell where X and Y do not share some subsequence, add that to Z for the shortest common supersequence. The key point is not to add redundant elements where both X and Y already share subsequences.

5.c. The shortest common subsequence of X = BARRACUDA and Y = ABRACADABRA is BARBRACUADABRA, with X taking precedence on the left.

5.d. Nontrivial Base cases:

X[a], Y[a]

Z = [a]. m+n-l = 1+1-1 = 1.

X[a], Y[b]

Z = [a,b]. m+n-0 = 1+1-0 = 2.

Inductive Cases:

X[...a, b], Y[...a, a]

Z = [...a, b, a]. 2 + 2 - 1 = 3

X[...a, b], Y[...a, b]

Z = [...a, b]. 2 + 2 - 2 = 2

... assumes the previous elements are already solved for shortest common subsequences.